Query Processing on Low-Energy Many-Core Processors

Annett Ungethüm[#], Dirk Habich[#], Tomas Karnagel[#], Wolfgang Lehner[#], Nils Asmussen[†] , Marcus Völp[†], Benedikt Nöthen[*], Gerhard Fettweis[*]

[#] *Database Technology Group, Technische Universität Dresden, Germany*
{annett.ungethuem, dirk.habich, tomas.karnagel,wolfgang.lehner}@tu-dresden.de

[†] *Operating Systems Group, Technische Universität Dresden, Germany*
{asmussen, voelp}@os.inf.tu-dresden.de

[*] *Vodafone Chair Mobile Communications Systems, Technische Universität Dresden, Germany*
{benedikt.noethen, gerhard.fettweis}@ifn.et.tu-dresden.de

***Abstract*—Aside from performance, energy efficiency is an increasing challenge in database systems. To tackle both aspects in an integrated fashion, we pursue a hardware/software co-design approach. To fulfill the energy requirement from the hardware perspective, we utilize a low-energy processor design offering the possibility to us to place hundreds to millions of chips on a single board without any thermal restrictions. Furthermore, we address the performance requirement by the development of several database-specific instruction set extensions to customize each core, whereas each core does not have all extensions. Therefore, our hardware foundation is a low-energy processor consisting of a high number of heterogeneous cores. In this paper, we introduce our hardware setup on a system level and present several challenges for query processing. Based on these challenges, we describe two implementation concepts and a comparison between these concepts. Finally, we conclude the paper with some lessons learned and an outlook on our upcoming research directions.**

I. Introduction

For the last 30 years, disk-centric systems based on commodity hardware exploiting only a minimal set of regular operating system services have reflected the state-of-the-art. Within the last years, however, this picture has dramatically changed due to several reasons, but especially due to significant developments in the hardware sector. The awareness of the need to be more focused on special capabilities of the underlying system, currently has a huge impact on research as well as on the commercial data management ecosystem. Moreover, the database community sparked a large number of extremely innovative research projects to push the envelope in the context of modern database system architectures. Using this spirit, our overall vision is to develop an energy-efficient database system using a hardware/software co-design approach. Therefore our research project is highly interdisciplinary involving researchers from the hardware sector, operating systems, and database systems.

Generally, in order to improve the energy efficiency of a database server, two independent directions can be pursued. On the one hand, the faster a query is being processed, the less energy is consumed [1]. For example, if a query can be answered using an index lookup instead of a table scan, fewer cycles are spent on that particular query. On the other hand, energy can be saved, if individual hardware components are turned off to save idle power and increase the utilization of running components. As a consequence, the individual response time of a query may suffer from improved energy efficiency. In this case, the system has to flexibly balance query response time minimization and throughput maximization under a given energy constraint on a case-by-case basis. In opposition to *elasticity in the large*, this property can be considered *elasticity in the small*.

In our research, we focus on the second direction of *elasticity in the small* by enhancing a single system by more and more components or cores, whereas we utilize low-energy processors. The extremely low-energy design offers us the possibility to place hundreds to millions of chips on a single board without any thermal restrictions. Fundamentally, general purpose processors are reaching their limits since single-threaded performance has almost stopped to increase, because the maximum core frequency is limited by physical constraints. Even the current solution, to put more and more homogeneous cores onto a single socket, will also reach physical limitations soon. As the feature size in which processors are manufactured will shrink, the number of transistors will increase, enlarging the occurrence of dark silicon [2]: Since not all transistors can be supplied with power at the same time, some fraction of the chip space can be used for additional specialized instruction sets to be power-gated whenever needed without compromising the overall general purpose characteristics of the chip itself. Thus, specialized circuits in form of instruction set extensions or heterogeneous cores are more helpful in terms of performance than placing more homogeneous cores on the chip.

In previous papers, we considered the aspect of heterogeneous cores by presenting specialized processor designs to efficiently support database system primitives. We tackled the challenge to optimize set-oriented database operations [3] and introduced specialized instruction set extensions for database hashing primitives [4]. Both design approaches are based on low-energy processors as described in our vision. In both papers, we clearly demonstrated the benefits of these specialized instruction set extensions in terms of performance as well as energy consumption compared to general purpose processors. In our ongoing research, we are going to develop

further extensions for other database primitives. This research direction enables us to build a low-energy chip consisting of heterogeneous low-energy cores specialized for database systems.

Our Contribution and Outline

Aside from the development of specialized instruction set extensions for the database system, we also have to tackle the challenge of query processing on a low-energy many-core processor. In this paper, we are going into a more system-oriented level and describe our approach to execute database queries on such a processor design. In detail, our contributions are as follows:

- At first, we introduce our underlying hardware architecture, the *Tomahawk* platform, which is developed at our university (Section II). Unfortunately, the current available prototype (*Tomahawk* 2 - T2) does not include any database-specific extension, which is pursued for the next release in 2015.
- Generally, the low-energy architecture design offers some benefits for query processing. However, the processing of large data sets is the most challenging part. In particular, the data transfer between the different cores is a major challenge, which is explained in Section III.
- Based on these challenges, we investigate and evaluate the current task execution concept of the *Tomahawk* platform for query processing in a database system. As we are going to show, the current available task concept is not sufficient for query processing.
- Based on the previous investigation, we describe and evaluate our alternative concept in Section V. This concept is implemented using an operating system for heterogeneous manycores.
- We conclude the paper with a description of our ongoing work in Section VI, a presentation of related work in Section VII and a brief summary in Section VIII.

II. Tomahawk Architecture

The hardware foundation of our hardware/software co-design is the *Tomahawk* platform [5]. This platform is a heterogeneous multiprocessor system-on-a-chip (MPSoC) and has been developed at our university, whereas the primary focus was on mobile communication applications. Nevertheless, the platform aims to be able to adapt for highly specialized tasks while being very energy efficient.

Generally, the Tomahawk platform consists of two subsystems called *control-plane* and *data-plane* [5]. The control-plane subsystem comprises a CPU, a global memory and peripherals, whereas this CPU is also called application core (App). The App-Core is responsible to execute the application control-flow. The data-plane subsystem consists of a number of processing elements (PEs), each equipped with a local program and data memory. PEs are not able to access the global memory directly, instead a data locality approach is exploited using scratch-pad local memory. That means, the PEs are explicitly isolated from the control-plane subsystem. The main purpose of this subsystem is to execute the pure data flow processing (accelerator approach for data processing). Therefore, this subsystem can be seen as slave unit in the overall system architecture. Both subsystem are decoupled logically and linked through a controller called CoreManager (CM). The CM is responsible for the task scheduling of the PEs, the PE allocation and the data transfer management from global memory to the PEs. Moreover, the CM performs frequency scaling of the PE cores in order to minimize the power consumption.

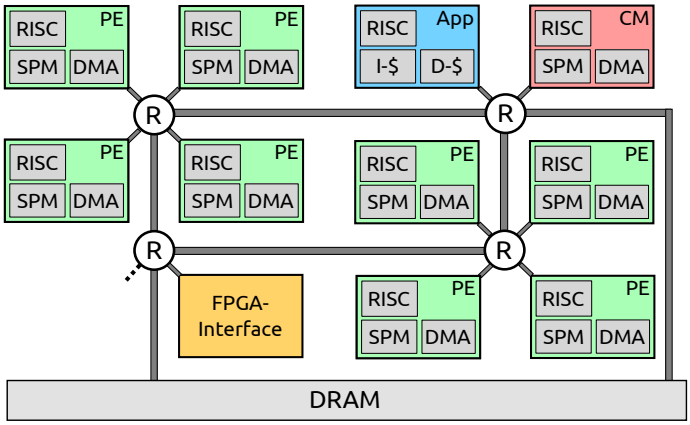


Figure 1: Simplified structure of the Tomahawk 2 chip.

In particular, figure 1 shows a conceptual representation of the *Tomahawk2* (T2) [6] including the elements of the control-plane and the data-plane. All modules are integrated onto the same die and are connected by a network-on-chip (NoC). Externally, there are two DRAM modules of 128MB each that are connected to the chip. The application core consists of a 570T core from Tensilica (now Cadence), and a 16 KB cache for code and data, each. In contrast to that, a PE also contains an Xtensa-LX4 core, but only 32KB scratchpad memory (SPM) for code and 32KB for data, i. e. no cache. The PEs can be clocked between 83 and 666 MHz. Additionally, it has a DMA unit that was originally intended for sending debug messages. However, it is general enough to allow arbitrary data transfers between all modules, which is utilized by M3 (see V). The CM contains a 32KB large scratchpad memory for code and 64KB scratchpad memory for data.

III. Query Processing Challenges and Opportunities

As described and shown in [5], the architecture is well-suited for mobile communications applications with stringent performance and energy-efficiency requirements, whereas each application consists of a control-flow description (e.g. using finite state machines) and a data-flow description (e.g. directed acyclic graph). In this case, each part is executed on the corresponding T2 subsystem orchestrated by the CoreManager.

From a conceptual perspective, query processing in database systems is similar to mobile communications applications, since each query is usually expressed as a directed acyclic graph. Therefore, the T2 concept should be benefical for that kind of application. Furthermore, the papers [3], [4] presented instruction set extensions for the PEs and demonstrated a comparable performance behaviour to general purpose processors with a very low energy consumption. Therefore, we investigated the whole T2 concept for query processing.

In contrast to mobile communication applications, the data aspect in database systems is more challenging. Fundamentally, the processing elements (PEs) are the main elements which are responsible for any computation. However, the PEs are hardware elements which are isolated by definition since they cannot interact. Each PE disposes of its own local scratchpad memory. That means, the PEs do not have direct access to the DRAM, but only to their own scratchpad memory. On the one hand, this prevents overhead due to the absence of a generic cache coherence protocol. On the other hand, this decreases the programmability, because the data flow needs to be handled explicitly and the directly accessible address space for the code running on a PE is only 32KB for code and 32KB for data. The size of the scratchpad memory will increase in future, but the aspect of isolated PEs will be kept. The challenge for query processing in database systems is that each PE is able to process only a limited size of data (e.g., 32KB) at once, which means that a fine-grained data partitioning is required and a lot of input and output transfers are issued. This raises the question, how the dataflow can be organized and managed efficiently. The concrete answer to this can vary for different use cases. However, the importance of this aspect grows, considering that the number of PEs is expected to increase significantly in the future, as the size of the data managed by database systems still increases.

The fine-grained partitioning and the high input and output transfer rate are challenging, but there exist some interesting and positive side effects for database systems. Based on the required fine-grained partitioning for query processing, each operator works per-se in intra-operator parallelism mode based on a task-level parallelism. In this way, the processing time for each operator task on fine-grained data partitions is reduced, so that we have to cope only with short running tasks. Using that short running task property, an elastic query processing can be established (*elasticity in the small* as mentioned in the introduction). That means, intra-operator, inter-operator and inter-query parallelism can be varied at any point in time, since we only have short-running tasks and a high flexibility by the assignment of tasks to PEs. The high flexibility results from the aspect, that we have to explicitly assign functions and fine-grained data partitions to the PEs in any case.

IV. Query Processing on Tomahawk

To establish a query processing for a database system on the T2 system, we have to do two things: (1) implementation of query operators and (2) constructing queries as data flow graphs. Furthermore, we have to consider the requirement of the fine-grained data partitioning as described in the previous section.

A. TaskC Programming Interface

Aside from the hardware concept, the paper [5] also introduces an appropriate programming interface called *TaskC* for short running tasks. TaskC is an extension for the C programming language allowing developers to code atomic tasks with input and output data, which are executable in parallel on the isolated PEs, whereas input and output data are of fixed lengths. An example is depicted in Listing 1. In this example, a function *task_doSomething* is defined that should be executed on a PE, taking an input array and an output array as arguments. The function body reads from the input array and fills thereby the output array. All task arguments must be pointers to arrays with an even length. Results are always written to one of these arrays. A direct transfer to the DRAM is not provided. Generally, the TaskC-concept is similar to CUDA kernels or OpenCL kernels. Therefore, the implementation of query operators is straightforward. Then, a database query is a data flow graph represented as host program and within this program, several tasks can be called, whereas we have to explicitly specify the parallelism by defining the number of tasks for each operator based on the data size. The size of input and output arrays must be stated. In the following example these sizes are constant but this is not necessary.

Listing 1: TaskC example code

```
void task_doSomething(int *in, int *out) {
    for (int i = 0; i < 16; i++)
        out[i] = in[i * 2] * in[i * 2 + 1];
}
int main() {
    task(task_doSomething,
        IN(inarr, 32 * 4), OUT(outarr, 16 * 4));
    taskSync();
}
```

B. Query Execution

To execute a query, the corresponding host program has to be loaded on the App-Core. Within this program, tasks can be called being executed on PEs. By specifying the input and output data for a task, the developer explicitly defines the dependencies of the tasks (data-flow approach). These dependencies are used by the *CoreManager* of the T2 system to schedule the tasks in the appropriate order on arbitrary PEs. For example, task A could use the output data of task B as the input data and would thereby create a dependency of task A on task B. Furthermore, the *CoreManager* also provides information about the data locality and delegates the data transfer which is then initiated by a control unit on the corresponding PE. All tasks are usually started asynchronously, therefore, an application can call *taskSync*, to synchronize the submitted tasks and wait for their completion.

C. Evaluation

To evaluate the performance of queries using TaskC, we implemented three applications executing common operators and measured their runtime behavior while changing the data size. All applications employ a single task function that executes the corresponding operator over a part of the data (fine-grained data partitioning). The first application performs a parallelized recursive bitonic sort, while the second performs an additional aggregation count. The third one does only perform the aggregation count. Figure 2a shows the results for the three applications. As expected, the aggregation is significantly faster than the sort and grows roughly linearly.

An important question for data-intensive applications is how expensive data transfers are. Therefore, we evaluated the amount of time spent with data transfers compared to the time spent with the actual computation. Figure 2b shows the runtime of a selection with three predicates. Every task

executes the whole selection including all predicates. The upper curve shows the complete time, while the intermediate curve excludes the transfer back to the DRAM. The time for the transfer back to the application core is illustrated by the lower curve. As we can see, the transfer time grows more than linearly when increasing the data size on the T2.

To further analyze the transfer time for this test case, we varied the number of the used PEs. Figure 2c shows the percentage of the time which is spent only for the result transfer. When using only a single PE it stays on a relatively constant level. However, it starts growing along with the data size when all PEs are used and finally levels out at a range between 17 and 20%.

D. Conclusion

Unfortunately, a query processing implementation with TaskC and execution using the proposed *CoreManager* suffers from the limitation that the resulting set of data has to be estimated before the task is executed producing unnecessarily large return arrays. In typical database systems, these are only known at runtime, when processing the input data. Furthermore, the whole output array is always sent back to the DRAM, regardless of the number of results that have actually been written. This leads to many expensive transfers which is especially bothering because typically, the resulting dataset is rather small, compared to the input dataset, e. g. for the reduce part of a Map Reduce.

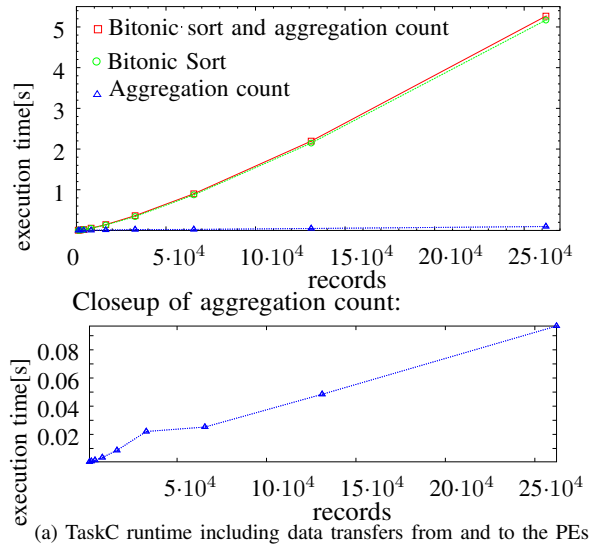
There are further restrictions regarding the different kinds of parallelism in database systems. While intra-operator and intra-query parallelism follow directly from the task concept, inter-operator/query parallelism is not acquirable. For instance, assume an operation depending on the results of two preceding operations, e.g. joining the outcome of two different select operations. The select operations would have to be started one after another and the join could only begin after all preceding tasks have been finished. Starting the different select operations at the same time on different assigned PEs or beginning the join before all preceding operations is part of the data locality optimization which is done by the Core Manager. The user can only influence this behaviour indirectly by adapting the working set. Additionally the Core Manager always saves a copy of the data to the main memory for failure reaction reasons.

V. Software-Controlled Query Processing on Tomahawk

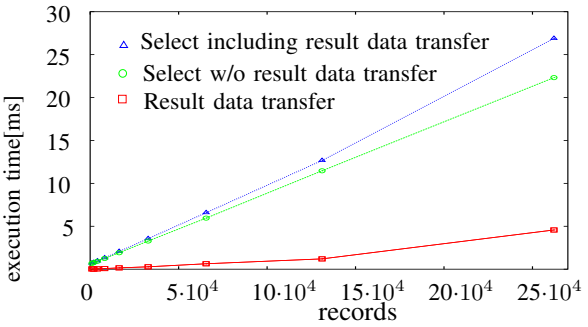
To overcome the previously described manual adaption of the working set for query processing, we have evaluated a microkernel-based operating system approach on the T2.

A. Mircokernel Approach on Tomahawk

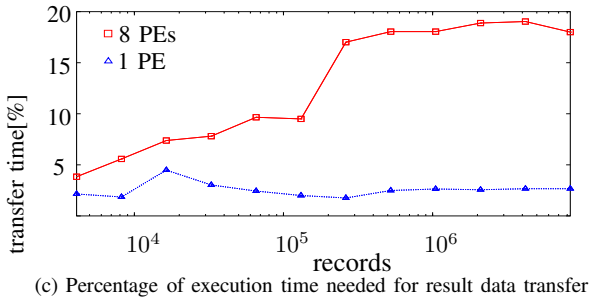
Our operation system group is developing an operating system (OS) called *M3* for heterogeneous manycore systems. Since the traditional approach of running a shared OS kernel on all cores will not work for heterogeneous cores, where some cores might not even have OS support, the concept of M3 is to run a microkernel on a dedicated core and remote-control the other cores. That is, both the microkernel and the applications run alone on their cores, whereas the



(a) TaskC runtime including data transfers from and to the PEs



(b) Select with 3 predicates, TaskC runtime



(c) Percentage of execution time needed for result data transfer

Figure 2: Runtimes using TaskC. Each record contains two 32-bit integer values

applications get linked against a library that provides them with abstractions for application creation, communication, and memory management.

In contrast to TaskC, where tasks are isolated by definition because they cannot communicate with other PEs or the DRAM, a central point of M3 is to increase the flexibility for applications by allowing them to communicate. However, allowing communication requires isolation, i.e. M3 needs to ensure that applications cannot influence or even destroy each other, because otherwise the system cannot be used by multiple applications at the same time. Since the microkernel is running on a dedicated core, it is not involved in a communication between applications, but data is directly exchanged between the cores the applications run on. For that reason, M3 builds upon a small hardware component for each core, that allows to establish communication channels between cores or between a core and a memory. This hardware component

can be used by the application, but its configuration, i.e. where data can be sent to, for example, is not accessible by the application, but only by the microkernel. Thereby, the communication capabilities of applications are in complete control of the microkernel (software-controlled processing).

M3 runs on multiple platforms. One of the supported platforms is Linux, which is used as a virtual machine to simulate the principle behavior of emerging hardware architectures that do not provide shared memory and have a hardware component that allows communication [7]. This can also be utilized as a development platform that allows quick iterations and has rich debugging support in contrast to T2, for example, which proved as very helpful for this work. Another supported platform is the T2. On the T2, a debug hardware component is used by M3 allowing the access to the DMA unit, mentioned earlier. It does not provide the isolation features yet because it allows a PE to communicate with every other PE. However, the concepts of M3 can still be applied, and the isolation is emulated by software checks on T2.

M3 allows the developer to run full applications on the PEs, instead of single functions. Over the microkernel, they can establish communication channels by one application registering itself as a service and other applications connecting to that service and acting thereby as a client. Another feature that M3 offers is the ability to allocate memory in DRAM and share that with other applications.

B. Evaluation

In contrast to TaskC, the M3 kernel and library allows event driven data transfer, i.e. data is only sent when necessary and the receiver implements an event handler for processing incoming messages. It may be expected that the application can profit from the absence of unnecessary transfers and the full control of the time data is left in the local memory.

In order to test this hypothesis, the scenario for M3 contains the same functionality as the TaskC test case, i.e. it selects records using three predicates. Figure 3 illustrates which programs are loaded onto the PEs. *Select 1* is called once for every part of the dataset while *Select 2* and *Select 3* are only called for those, which have passed the first select operator. The first operator is expected to narrow the result set and therefore reduce the workload for the succeeding process. Of course, all predicates could be tested on the same PEs. This would accelerate the execution even more. But the goal of this specific comparison is to detect differences in the data transfer time.

Fig. 4 shows the overall runtime of the process for different selectivities and the runtime for TaskC using 4 PEs for comparison. There is no differentiation between the selectivities for TaskC, because it does not change due to the static result array size. For smaller selectivities, the M3-based approach is the more efficient one. Although it only uses two PEs at once per operator while TaskC uses four of them and the application core to host the control program. For larger selectivities, the benefit of the M3-based approach shrinks because more data needs to be transferred to *Select 2 & 3* and more computation on these PEs is required. However, due to the overhead of dynamic data dependency checking in the TaskC approach, the M3-based approach is still slightly

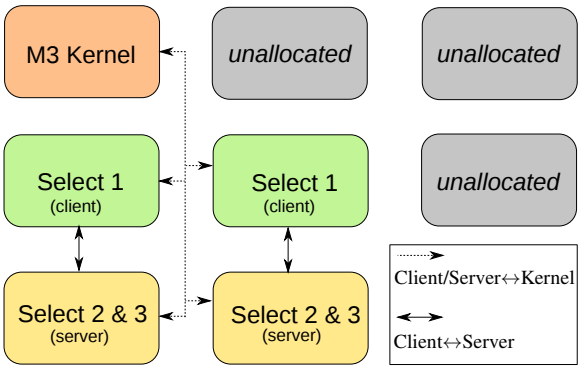


Figure 3: Configuration of the eight processing elements

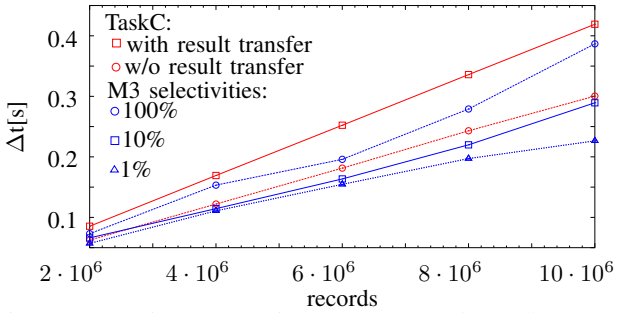


Figure 4: Runtime comparison between TaskC and M3. TaskC runtimes are not differentiated by selectivity since the amount of transferred data is always the same.

faster with 100% selectivity. By comparing the results of M3 with 1% selectivity and the TaskC approach without result transfers, it can also be seen that the missing opportunity to specify the actual size of the results decreases the performance significantly, but still does not reach the performance level of M3.

Because of the large differences of the approaches, the comparison is difficult though. For example, in the TaskC version all three operators are applied by one task, instead of applying two separate applications, as with M3. This leads to duplicate work. Moreover, M3 needs to emulate missing hardware functionality, which adds some overhead that is more severe the more transfers are done, i.e. the more data is selected in the first step.

Attention should be paid to the fact that the programs implementing the operators can be loaded on as many cores as the developer wishes. Moreover, different programs can be loaded. In addition to intra-operator/query parallelism this allows inter-operator/query parallelism.

VI. Lessons Learned

As shown, there are different approaches to implement query processing functionalities on the T2 platform. On the one hand, *TaskC* uses a concept of short running tasks with the same functionality being automatically distributed over a number of cores that is already widely used, e.g. in CUDA or OpenCL. The most recognizable downside is the occasionally large transfer overhead and the limitation, that only arrays can be passed to a task. The full potential of TaskC is tapped when executing many short running tasks which have no need for communication between each other, e. g. in mobile communication for which the Tomahawk was initially developed.

Database applications using TaskC would benefit from DRAM access and a possibility to communicate directly between the PEs to avoid the long route via the DRAM. Additionally, specific database functionality, e.g. query partitioning, could be a part of the *CoreManager*.

On the other hand, the software-controlled approach M3 allows the developer to define separate queries or operators for the PEs and perform data transfers between them. This reduces unnecessary data transfers to a minimum and provides the developer with a much higher degree of freedom in choices concerning parallel implementations. Nevertheless, queries using M3 have a more expensive setup time compared to starting a task on a PE when using TaskC, because they need to establish communication capabilities over the kernel first. That means, for short running programs (e.g. a few thousand cycles), TaskC is the more suitable approach.

On the hardware side, query processing would not only benefit from specific instructions, like already done [4], but also by a ring buffer for the DMA unit, which is planned for a later release of the Tomahawk. The ring buffer would allow data streaming and could thereby further speed up data intensive applications. Enforcing isolation in the DMA unit would relieve M3 from emulating the security checks in software and thus decrease the overhead for communication, which is also planned for a later release of the Tomahawk. Moreover, larger scratchpad memories would increase the amount of data that can be processed without requiring additional transfers, which would increase the performance.

VII. Related Work

In the last recent years, GPUs have been more and more used as database co-processors [8]. GPUs consist of many small cores, however, these cores cannot execute instructions independently of each other, but rather run the same operations highly parallel on different data. This introduces the burden of adjusting the algorithm to this specific kind of parallelism. On the T2, an algorithm can be implemented as Task or as PE code for M3, where it is executed as a single threaded program. Parallelism is added by executing many independent program parts at the same time.

More comparable to the T2 platform are the IBM Cell processor and the Intel Xeon Phi. The first has been evaluated as a database processor [9], [10]. It consists of eight small cores and one larger PowerPC core, which has a configuration very similar to the T2. Intels Xeon Phi has up to 61 small x86 cores, which have been used for database algorithms [11] including Map Reduce [12]. In contrast, the T2 additionally contains a core manager, which can be used for fast task processing, including dependency checks and task distribution. Using TaskC together with M3 allows two programming methods that can coexist, which is not given by the Cell architecture or the Xeon Phi. Also, the T2 is optimized for ultra low energy consumption far below x86 or Cell cores.

VIII. Conclusion and Outlook

We elaborated the challenges posed by a low-energy multicore architecture and investigated two different approaches for overcoming them. Basically they implement a static and a dynamic method for applying data flow graphs. The first one, implemented using TaskC, supports fault tolerance and dependency checking. While these features are very welcome, they also produce unwanted overhead and restrict the developer. The latter, implemented using M3, lacks these features but also the overhead. Additionally it provides the developer with more freedom. Consequentially we suggested extensions which are useful in terms of data processing. Hereafter, implementing them is a meaningful part of our work.

Due to the ongoing development of the Tomahawk, there will be new hardware releases featuring a wider palette of different cores, requiring further tests with the aid of the newly added or improved modules. This will show which strategies are optimal for different kinds of scenarios, clearing the way not only for more complex operations but also for implementing appropriate database libraries and frameworks, e. g. a Map Reduce.

IX. Acknowledgments

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and by the European Union together with the Free State of Saxony through the ESF young researcher group "IMData" 100098198.

References

[1] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *SIGMOD Conference*, 2010, pp. 231–242.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[3] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner, "An application-specific instructions set for accelerating set-oriented database primitives," in *SIGMOD*, 2014.

[4] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel, and W. Lehner, "Hashi: An application specific instruction set extension for hashing." in *ADMS@VLDB*, 2014, pp. 25–33.

[5] O. Arnold, E. Matus, B. Noethen, M. Winter, T. Limberg, and G. Fettweis, "Tomahawk: Parallelism and heterogeneity in communications signal processing mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 107, 2014.

[6] B. Noethen, O. Arnold, E. Perez Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matus, G. Fettweis, H. Eisenreich, G. Ellguth, S. Hartmann, S. Hoppner, S. Schiefer, J.-U. Schlusler, S. Scholze, D. Walter, and R. Schuffny, "10.7 a 105gops 36mm2 heterogeneous sdr mpsoc with energy-aware dynamic scheduling and iterative detection-decoding for 4g in 65nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, Feb 2014, pp. 188–189.

[7] N. Asmussen, H. Härtig, and M. Völp, "Turning x86 into a hardware simulator for future manycores," in *Proceedings of the 3rd Workshop on Systems for Future Multicore Architectures*, Apr. 2013.

[8] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 21:1–21:39, Dec. 2009.

[9] M. de Kruijf and K. Sankaralingam, "Mapreduce for the cell broadband engine architecture," *IBM J. Res. Dev.*, vol. 53, no. 5, pp. 747–758, Sep. 2009.

[10] B. Gedik, R. R. Bordawekar, and P. S. Yu, "Celljoin: A parallel stream join operator for the cell processor," *The VLDB Journal*, vol. 18, no. 2, pp. 501–519, Apr. 2009.

[11] B. Schlegel, T. Karnagel, T. Kiefer, and W. Lehner, "Scalable frequent itemset mining on many-core processors," in *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, ser. DaMoN '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:8.

[12] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh, "Optimizing the mapreduce framework on intel xeon phi coprocessor." in *BigData Conference*, X. Hu, T. Y. Lin, V. Raghavan, B. W. Wah, R. A. Baeza-Yates, G. Fox, C. Shahabi, M. Smith, Q. Y. 0001, R. Ghani, W. Fan, R. Lempel, and R. Nambiar, Eds. IEEE, 2013, pp. 125–130.